Lecture 23: Distributed Memory Machines and Programming

Concurrent and Multicore Programming
CSE 436/536

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan

Topics (Part 2)

- Parallel architectures and hardware
  - Parallel computer architectures
  - **Memory hierarchy and cache coherency**
- Manycore GPU architectures and programming
  - **GPUs architectures**
  - **CUDA programming**
- ☛ Programming on large scale systems (Chapter 6)
  - **MPI (point to point and collectives)**
  - Introduction to PGAS languages, UPC and Chapel
- Parallel algorithms (Chapter 8,9 &10)
  - **Dense matrix, and sorting**

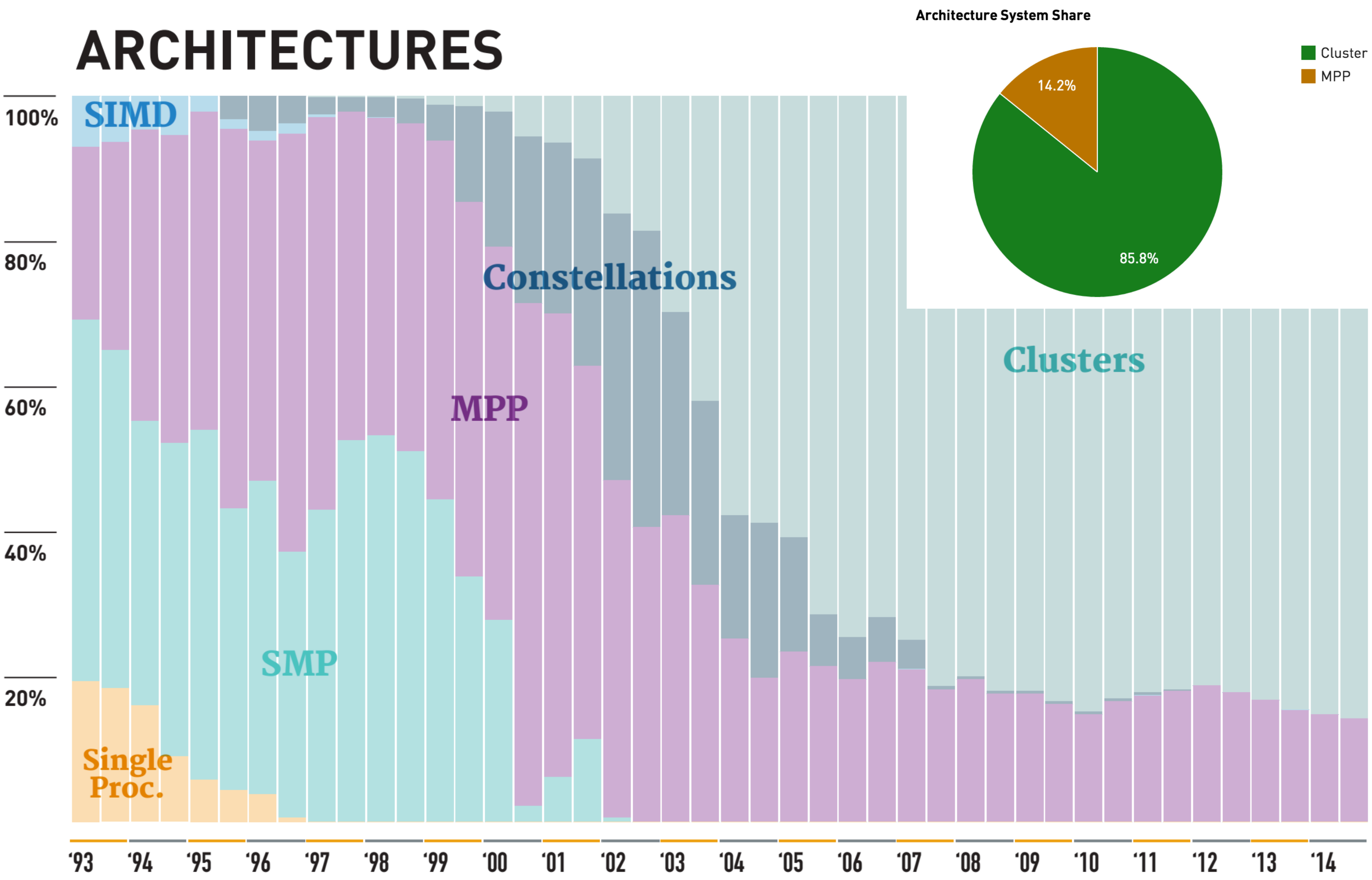
Acknowledgement

- Slides adapted from U.C. Berkeley course CS267/EngC233 Applications of Parallel Computers by Jim Demmel and Katherine Yelick, Spring 2011
  - http://www.cs.berkeley.edu/~demmel/cs267_Spr11/
- And materials from various sources

Recap: Node-level Architecture and Programming

- Shared memory multiprocessors: multicore, SMP, NUMA
  - Deep memory hierarchy, distant memory much more expensive to access.
  - Machines scale to 10s or 100s of processors
  - Instruction Level Parallelism (ILP), Data Level Parallelism (DLP) and Thread Level Parallelism (TLP)
  - **OpenMP, Cilk, pthread, etc**
- Manycore and heterogeneous system
  - Discrete memory space between host and accelerators
  - Manycore goes to 1000s PUs
  - SIMT or other type of lightweight threading model
  - **CUDA, OpenMP/OpenACC, etc**

HPC Architectures (TOP500, Nov 2014)



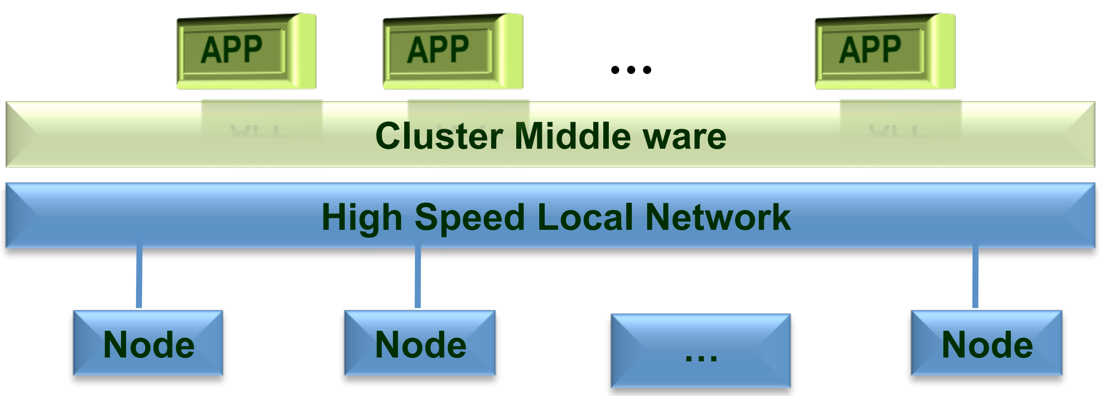
Outline

☛ Cluster Introduction

- Distributed Memory Architectures
  - Properties of communication networks
  - Topologies
  - Performance models
- Programming Distributed Memory Machines using Message Passing
  - Overview of MPI
  - Basic send/receive use
  - Non-blocking communication
  - Collectives

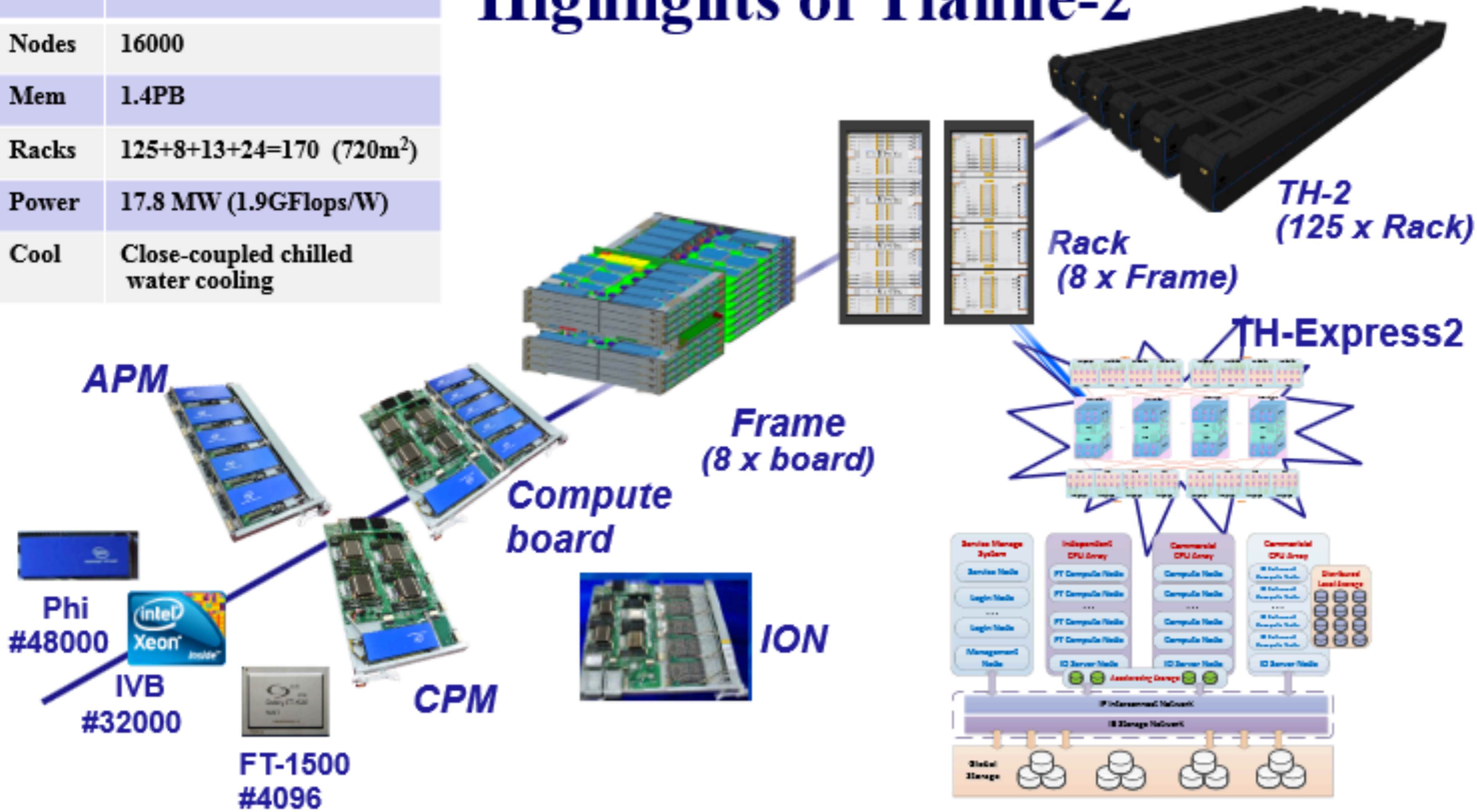
Clusters

- A group of linked computers, working together closely so that in many respects they form a single computer.
- Consists of
  - Nodes(Front + computing)
  - Network
  - Software: OS and middleware



Perf	54.9PFlops / 33.86PFlops
Nodes	16000
Mem	1.4PB
Racks	125+8+13+24=170 (720m²)
Power	17.8 MW (1.9GFlops/W)
Cool	Close-coupled chilled water cooling

Highlights of Tianhe-2



Hybrid Hierarchy shared storage System
H^2FS 12.4PB

#1 of Top500 Released 06/20/2016

New Chinese Supercomputer Named World's Fastest System on Latest TOP500 List

June 20, 2016, 4:01 a.m.

FRANKFURT, Germany; BERKELEY, Calif.; and KNOXVILLE, Tenn.—China maintained its No. 1 ranking on the 47th edition of the TOP500 list of the world's top supercomputers, but with a new system built entirely using processors designed and made in China. Sunway TaihuLight is the new No. 1 system with 93 petaflop/s (quadrillions of calculations per second) on the LINPACK benchmark.

Read more

http://www.top500.org/

NEWS

China Tops Supercomputer Rankings with New 93-Petaflop Machine

Michael Feldman, June 20, 2016, 9 a.m.

A new Chinese supercomputer, the Sunway TaihuLight, captured the number one spot on the latest TOP500 list of supercomputers released on Monday morning at the International Supercomputing Conference (ISC) being held in Frankfurt, Germany. With a Linpack mark of 93 petaflops, the system outperforms the former

IN DEPTH

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWE (KW)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
4	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
5	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660

China Races Ahead in TOP500 Supercomputer List, Ending US Supremacy

http://www.top500.org/lists/2016/06/

Top 10 of Top500

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
4	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
5	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
6	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
7	DOE/NNSA/LANL/SNL United States	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	301,056	8,100.9	11,078.9	
8	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
9	HLRS - Höchstleistungsrechenzentrum Stuttgart Germany	**Hazel Hen** - Cray XC40, Xeon E5-2680v3 12C 2.5GHz, Aries interconnect Cray Inc.	185,088	5,640.2	7,403.5	
10	King Abdullah University of Science and Technology Saudi Arabia	**Shaheen II** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	196,608	5,537.0	7,235.2	2,834

Cluster Classification

High availability clusters (HA) (Linux)

- Mission critical applications
- Also known as Failover Clusters, implemented for the purpose of improving the availability of services which the cluster provides.
- provide redundancy
- eliminate single points of failure.

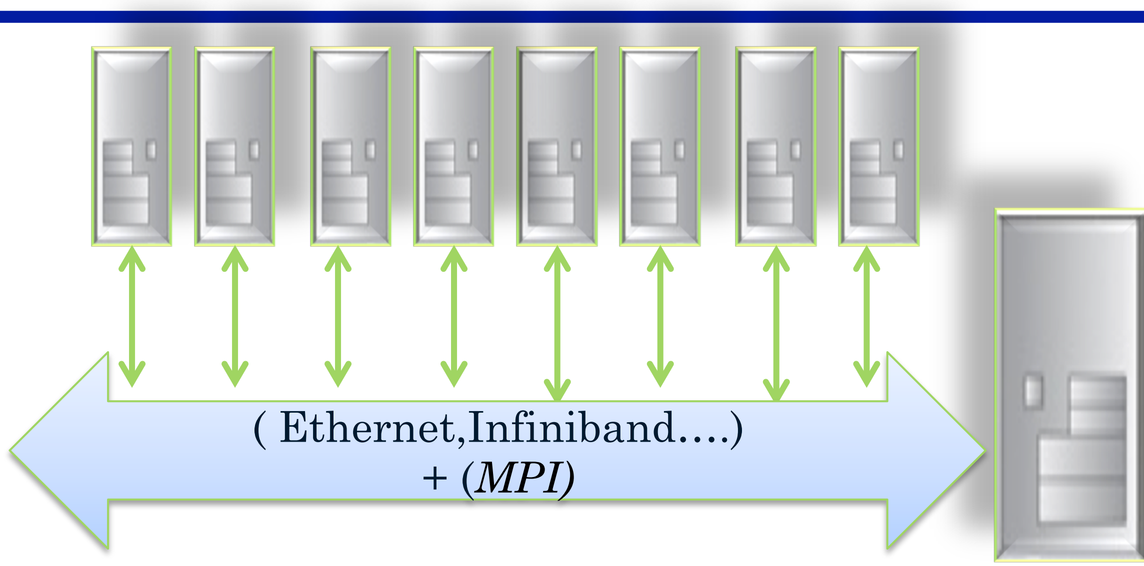
Network Load balancing clusters

- operate by distributing a workload evenly over multiple back end nodes.
- Typically the cluster will be configured with multiple redundant load-balancing front ends.
- all available servers process requests.
- Web servers, mail servers,..

HPC Clusters

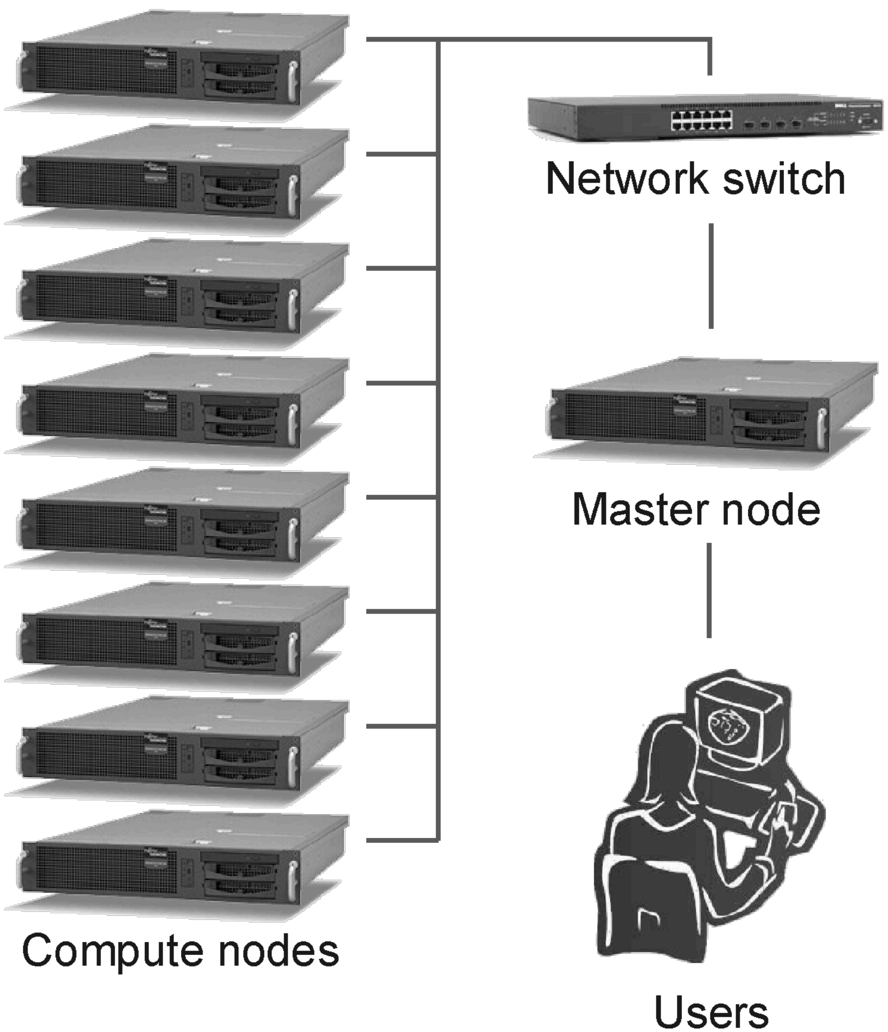
- Aims for high performance and throughput
- High-speed inter-connect
- Beowulf

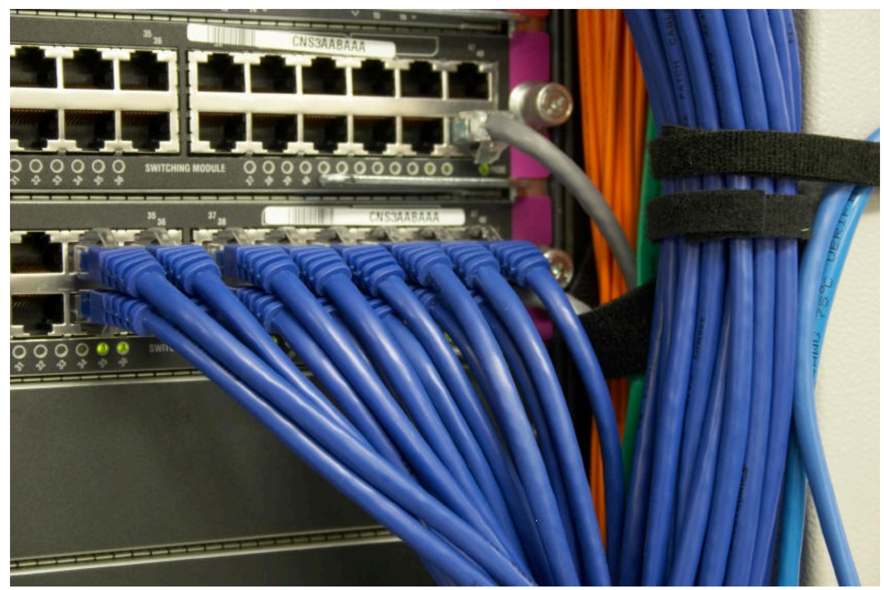
HPC Beowulf Cluster



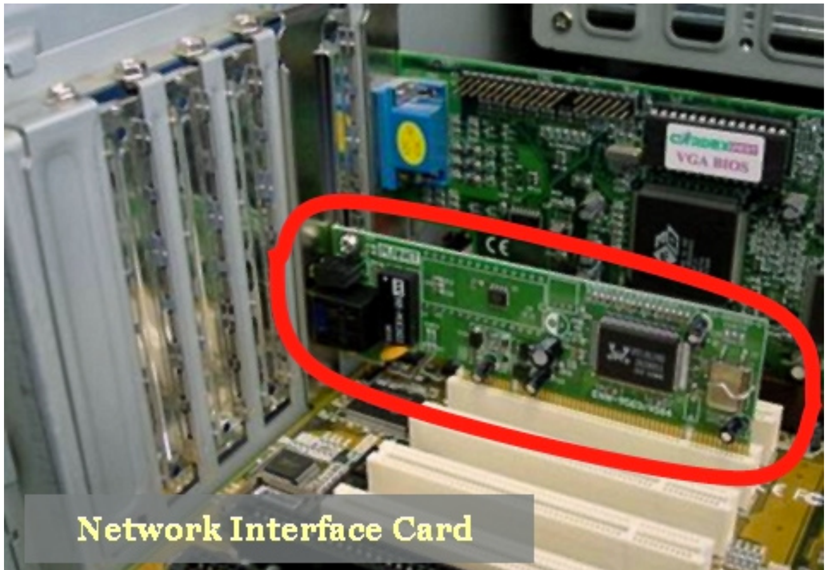
- Master node: or service/front node (used to interact with users locally or remotely)
- Computing Nodes : performance computations
- Interconnect and switch between nodes: e.g. G/10G-bit Ethernet, Infiniband
- Inter-node programming
  - MPI(Message Passing Interface) is the most commonly used one.

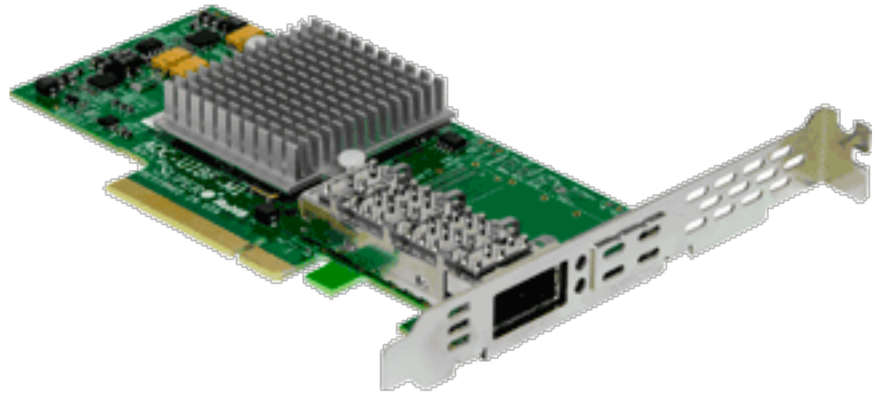
Network Switch

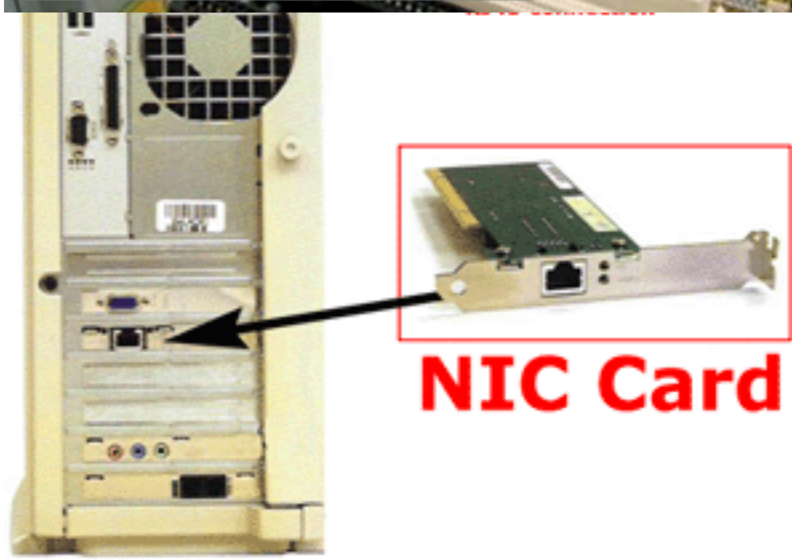




Network Interface Card (NIC)





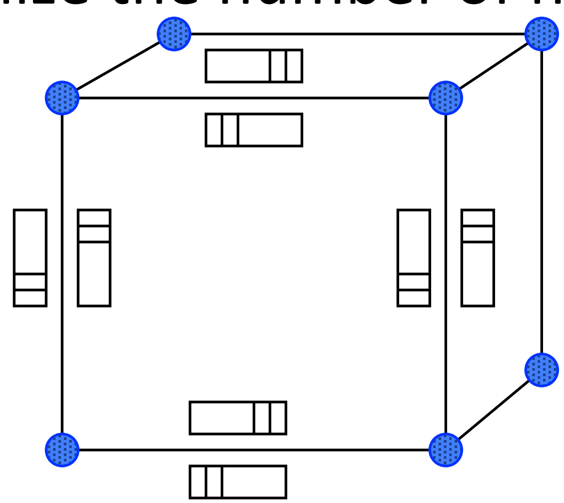


Outline

- Cluster Introduction
- ☛ Distributed Memory Architectures
  - Properties of communication networks
  - Topologies
  - Performance models
- Programming Distributed Memory Machines using Message Passing
  - Overview of MPI
  - Basic send/receive use
  - Non-blocking communication
  - Collectives

Historical Perspective

- Early distributed memory machines were:
  - Collection of microprocessors.
  - Communication was performed using bi-directional queues between nearest neighbors.
- Messages were forwarded by processors on path.
  - "Store and forward" networking
- There was a strong emphasis on topology in algorithms, in order to minimize the number of hops = minimize time



Network Analogy

- To have a large number of different transfers occurring at once, you need a large number of distinct wires
  - Not just a bus, as in shared memory
- Networks are like streets:
  - Link = street.
  - Switch = intersection.
  - Distances (hops) = number of blocks traveled.
  - Routing algorithm = travel plan.
- Properties:
  - **Latency: how long to get between nodes in the network.**
  - **Bandwidth: how much data can be moved per unit time.**
    - **Bandwidth is limited by the number of wires and the rate at which each wire can accept data.**

Latency and Bandwidth

- Latency: Time to travel from one location to another for a vehicle
  - Vehicle type (large or small messages)
  - Road/traffic condition, speed-limit, etc
- Bandwidth: How many cars and how fast they can travel from one location to another
  - Number of lanes

Design Characteristics of a Network

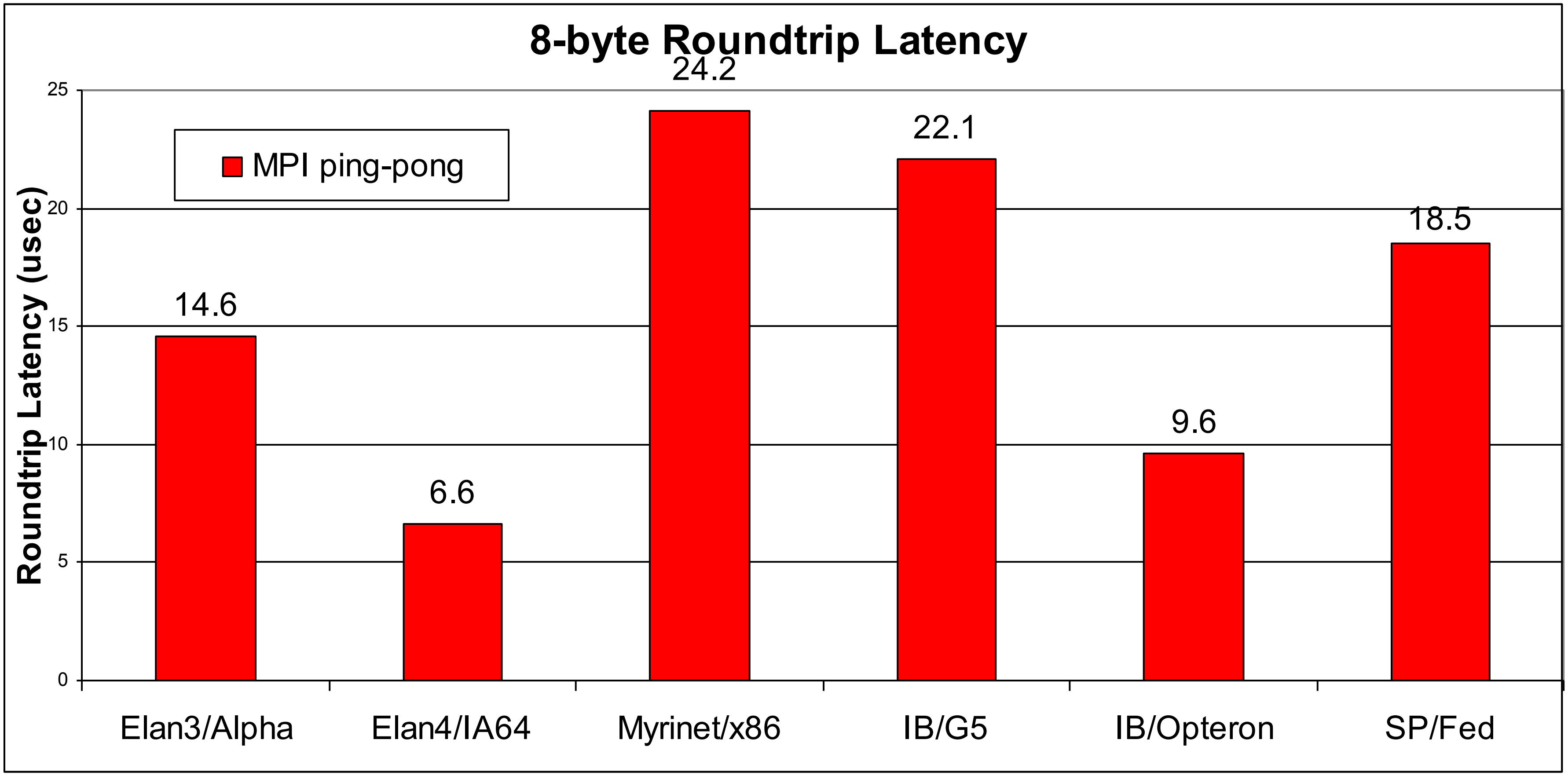
- Topology (how things are connected)
  - Crossbar, ring, 2-D and 3-D mesh or torus, hypercube, tree, butterfly, perfect shuffle ....
- Routing algorithm:
  - Example in 2D torus: all east-west then all north-south (avoids deadlock).
- Switching strategy:
  - Circuit switching: full path reserved for entire message, like the telephone.
  - Packet switching: message broken into separately-routed packets, like the post office.
- Flow control (what if there is congestion):
  - Stall, store data temporarily in buffers, re-route data to other nodes, tell source node to temporarily halt, discard, etc.

Performance Properties of a Network: Latency

- Diameter: the maximum (over all pairs of nodes) of the shortest path between a given pair of nodes.
- Latency: delay between send and receive times
  - Latency tends to vary widely across architectures
  - Vendors often report hardware latencies (wire time)
  - Application programmers care about software latencies (user program to user program)
- Observations:
  - Latencies differ by 1-2 orders across network designs
  - Software/hardware overhead at source/destination dominate cost (1s-10s usecs)
  - Hardware latency varies with distance (10s-100s nsec per hop) but is small compared to overheads
- **Latency is key for programs with many small messages**

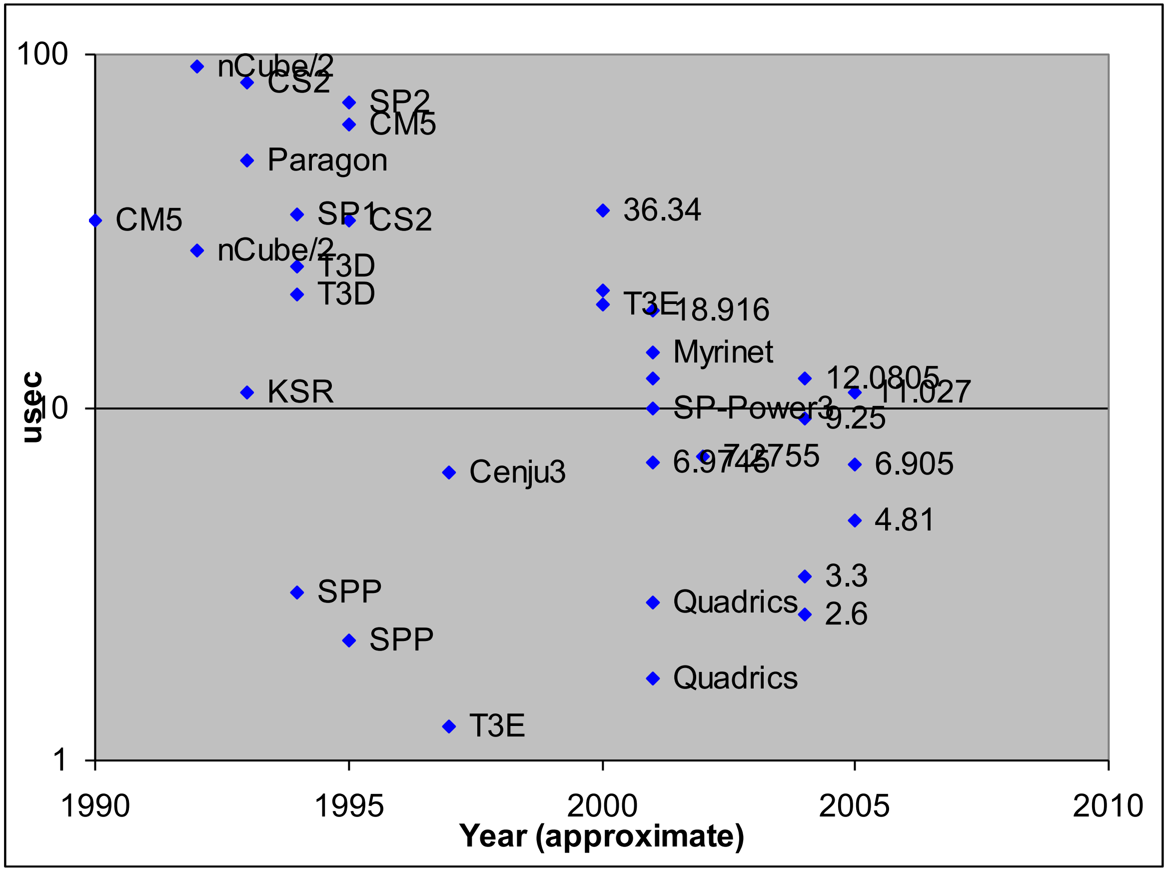
I second = 10^3 millseconds (ms) = 10^6 microseconds (us) = 10^9 nanoseconds (ns)

Latency on Some Machines/Networks



- Latencies shown are from a ping-pong test using MPI
- These are roundtrip numbers: many people use ½ of roundtrip time to approximate 1-way latency (which can't easily be measured)

End to End Latency (1/2 roundtrip) Over Time



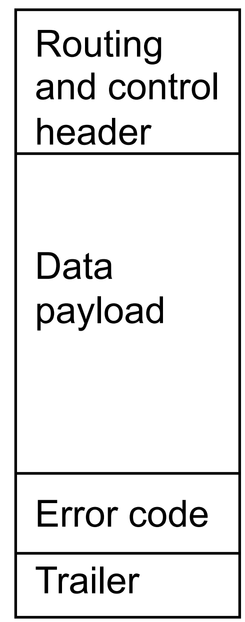
- Latency has not improved significantly, unlike Moore's Law
  - T3E (shmem) was lowest point – in 1997

Data from Kathy Yelick, UCB and NERSC

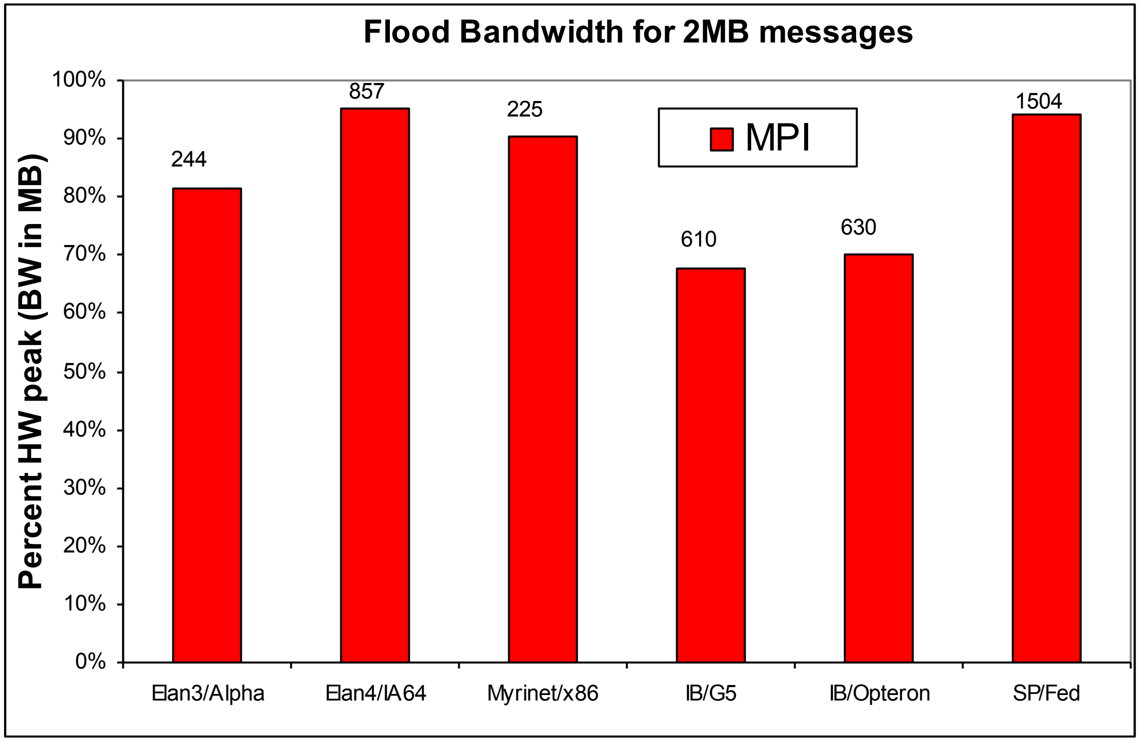
Performance Properties of a Network: Bandwidth

- The bandwidth of a link = # wires / time-per-bit
- Bandwidth typically in Gigabytes/sec (GB/s), i.e., 8* 2^{20} bits per second
- Effective bandwidth is usually lower than physical link bandwidth due to packet overhead.

- **Bandwidth is important for applications with mostly large messages**

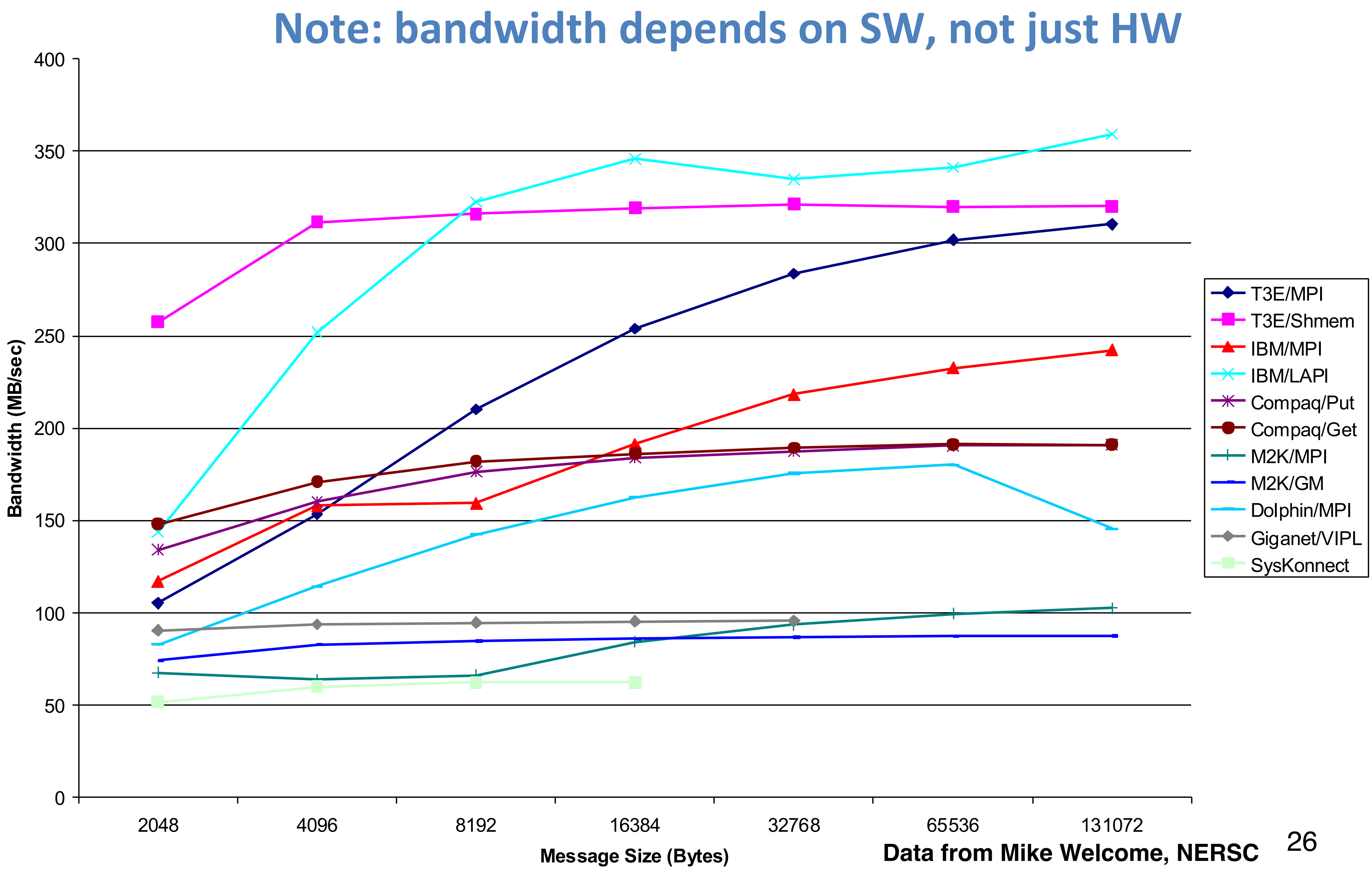


Bandwidth on Existing Networks



- Flood bandwidth (throughput of back-to-back 2MB messages)

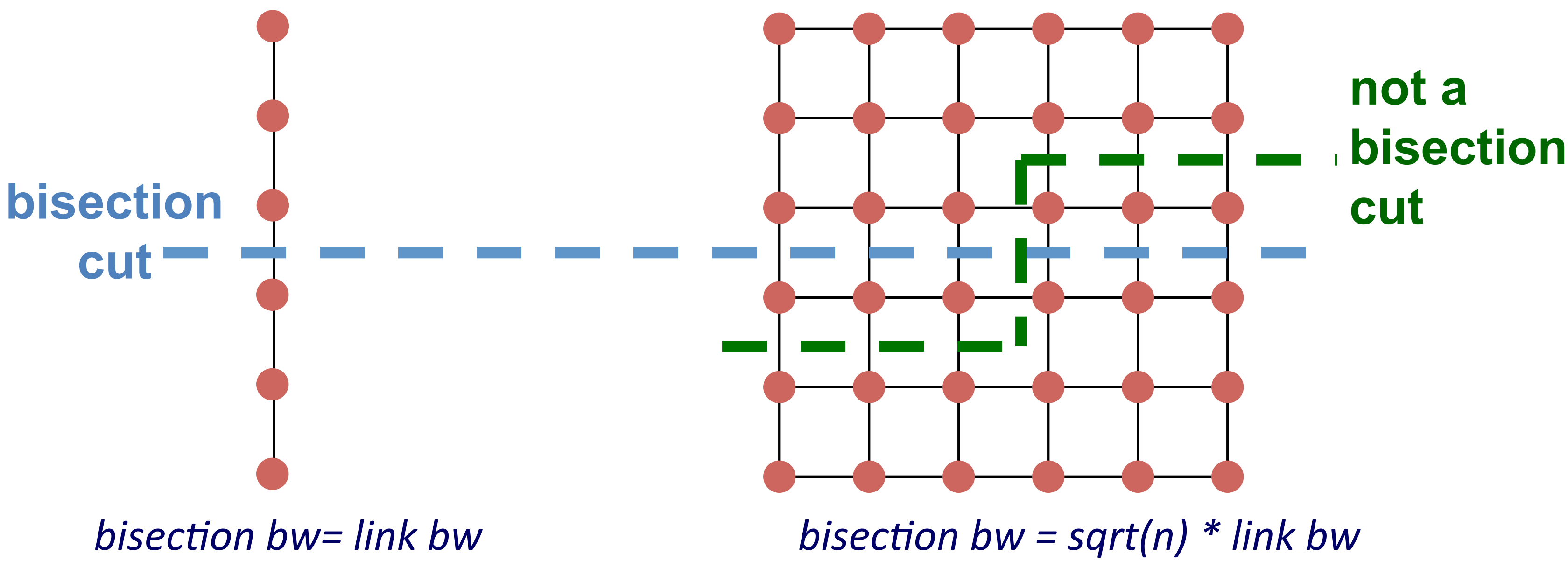
Bandwidth Chart



Data from Mike Welcome, NERSC

Performance Properties of a Network: Bisection Bandwidth

- Bisection bandwidth: bandwidth across smallest cut that divides network into two equal halves
- Bandwidth across "narrowest" part of the network



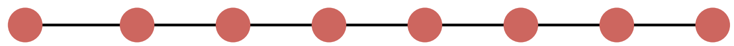
- Bisection bandwidth is important for algorithms in which all processors need to communicate with all others

Network Topology

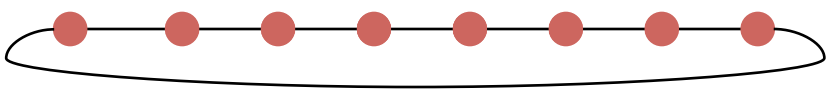
- In the past, there was considerable research in network topology and in mapping algorithms to topology.
  - Key cost to be minimized: number of "hops" between nodes (e.g. "store and forward")
  - Modern networks hide hop cost (i.e., "wormhole routing"), so topology is no longer a major factor in algorithm performance.
- Example: On IBM SP system, hardware latency varies from 0.5 usec to 1.5 usec, but user-level message passing latency is roughly 36 usec.
- Need some background in network topology
  - Algorithms may have a communication topology
  - Topology affects bisection bandwidth.

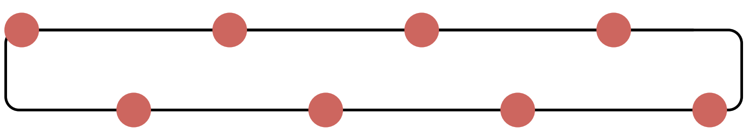
Linear and Ring Topologies

- Linear array



  - Diameter = n-1; average distance ~n/3.
  - Bisection bandwidth = 1 (in units of link bandwidth).
- Torus or Ring

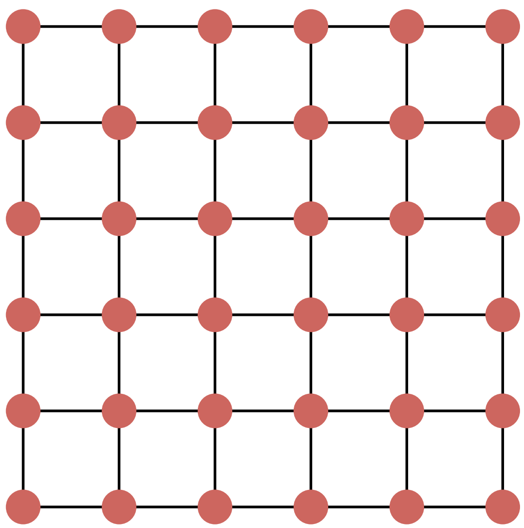




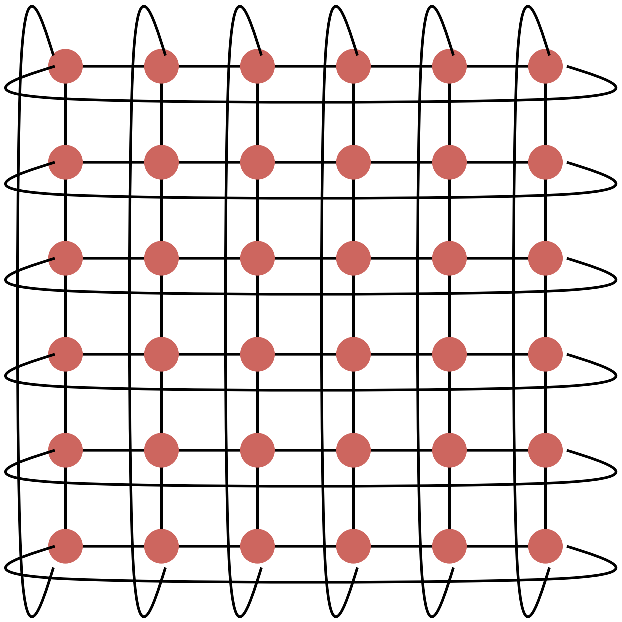
  - Diameter = n/2; average distance ~ n/4.
  - Bisection bandwidth = 2.
  - Natural for algorithms that work with 1D arrays.

Meshes and Tori

- Two dimensional mesh
  - Diameter = 2 * (sqrt(n) – 1)
  - Bisection bandwidth = sqrt(n)



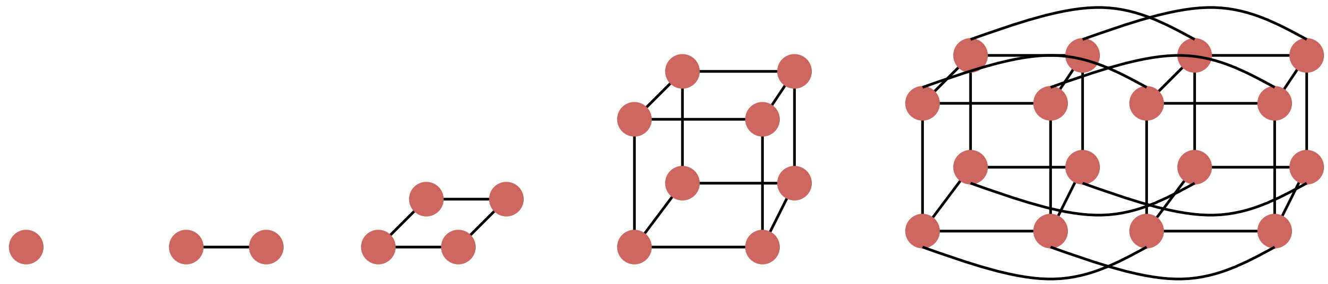
- Two dimensional torus
  - Diameter = sqrt(n)
  - Bisection bandwidth = 2* sqrt(n)



- **Generalizes to higher dimensions**
  - **Cray XT (eg Franklin@NERSC) uses 3D Torus**
- **Natural for algorithms that work with 2D and/or 3D arrays (matmul)**

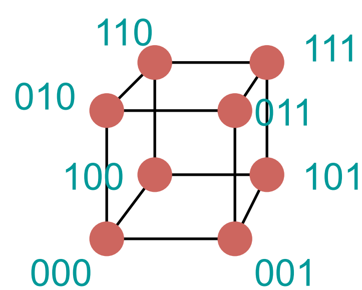
Hypercubes

- Number of nodes n = 2^d for dimension d.
  - Diameter = d.
  - Bisection bandwidth = n/2.



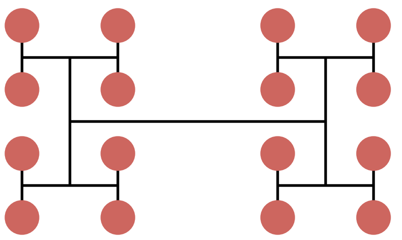
- 0d 1d 2d 3d 4d

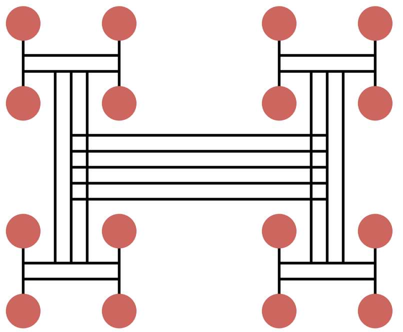
- Popular in early machines (Intel iPSC, NCUBE).
  - Lots of clever algorithms.
- Greycode addressing:
  - Each node connected to others with 1 bit different.

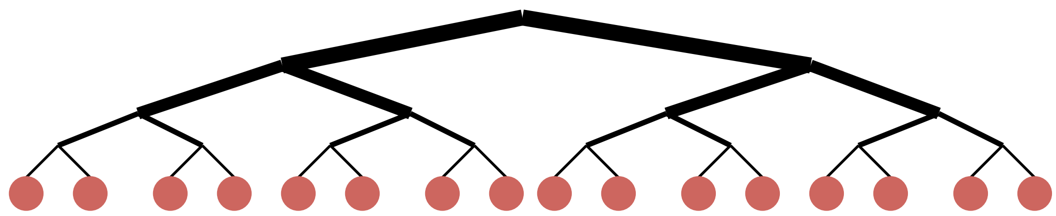


Trees

- Diameter = log n.
- Bisection bandwidth = 1.
- Easy layout as planar graph.
- Many tree algorithms (e.g., summation).
- Fat trees avoid bisection bandwidth problem:
  - More (or wider) links near top.
  - Example: Thinking Machines CM-5.



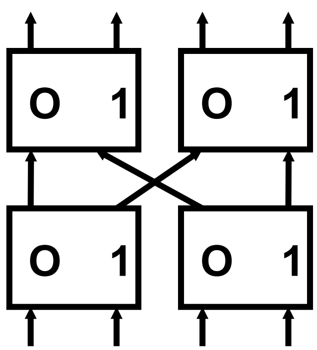




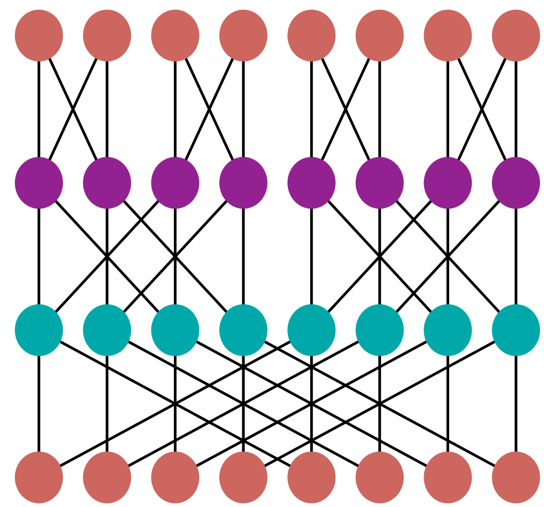
Butterflies

- Diameter = log n.
- Bisection bandwidth = n.
- Cost: lots of wires.
- Used in BBN Butterfly.
- Natural for FFT.

Ex: to get from proc 101 to 110,
Compare bit-by-bit and
Switch if they disagree, else not



butterfly switch



multistage butterfly network

Topologies in Real Machines

newer
older

Cray XT3 and XT4	3D Torus (approx)
Blue Gene/L	3D Torus
SGI Altix	Fat tree
Cray X1	4D Hypercube*
Myricom (Millennium)	Arbitrary
Quadrics (in HP Alpha server clusters)	Fat tree
IBM SP	Fat tree (approx)
SGI Origin	Hypercube
Intel Paragon (old)	2D Mesh
BBN Butterfly (really old)	Butterfly

Many of these are approximations:
E.g., the X1 is really a "quad bristled hypercube" and some of the fat trees are not as fat as they should be at the top

Evolution of Distributed Memory Machines

- Special queue connections are being replaced by direct memory access (DMA):
  - Processor packs or copies messages.
  - Initiates transfer, goes on computing.
- Wormhole routing in hardware:
  - Special message processors do not interrupt main processors along path.
  - Long message sends are pipelined.
  - Processors don't wait for complete message before forwarding
- Message passing libraries provide store-and-forward abstraction:
  - Can send/receive between any pair of nodes, not just along one wire.
  - Time depends on distance since each processor along path must participate.

Performance Models

Shared Memory Performance Models

- Parallel Random Access Memory (PRAM)
- All memory access operations complete in one clock period -- no concept of memory hierarchy ("too good to be true").
  - OK for understanding whether an algorithm has enough parallelism at all.
  - Parallel algorithm design strategy: first do a PRAM algorithm, then worry about memory/communication time (sometimes works)
- Slightly more realistic versions exist
  - E.g., Concurrent Read Exclusive Write (CREW) PRAM.
  - Still missing the memory hierarchy

Latency and Bandwidth Model

- Time to send message of length n is roughly

Time = latency + n*cost_per_word
= latency + n/bandwidth

- Topology is assumed irrelevant.
- Often called "α-β model" and written

$$\text{Time} = \alpha + n*\beta$$

- Usually $\alpha >> \beta >>$ time per flop.
  - One long message is cheaper than many short ones.

$$\alpha + n*\beta << n*(\alpha + 1*\beta)$$

  - Can do hundreds or thousands of flops for cost of one message.
- Lesson: Need large computation-to-communication ratio to be efficient.

Alpha-Beta Parameters on Current Machines

- These numbers were obtained empirically

machine	α	β
T3E/Shm	1.2	0.003
T3E/MPI	6.7	0.003
IBM/LAPI	9.4	0.003
IBM/MPI	7.6	0.004
Quadrics/Get	3.267	0.00498
Quadrics/Shm	1.3	0.005
Quadrics/MPI	7.3	0.005
Myrinet/GM	7.7	0.005
Myrinet/MPI	7.2	0.006
Dolphin/MPI	7.767	0.00529
Giganet/VIPL	3.0	0.010
GigE/VIPL	4.6	0.008
GigE/MPI	5.854	0.00872

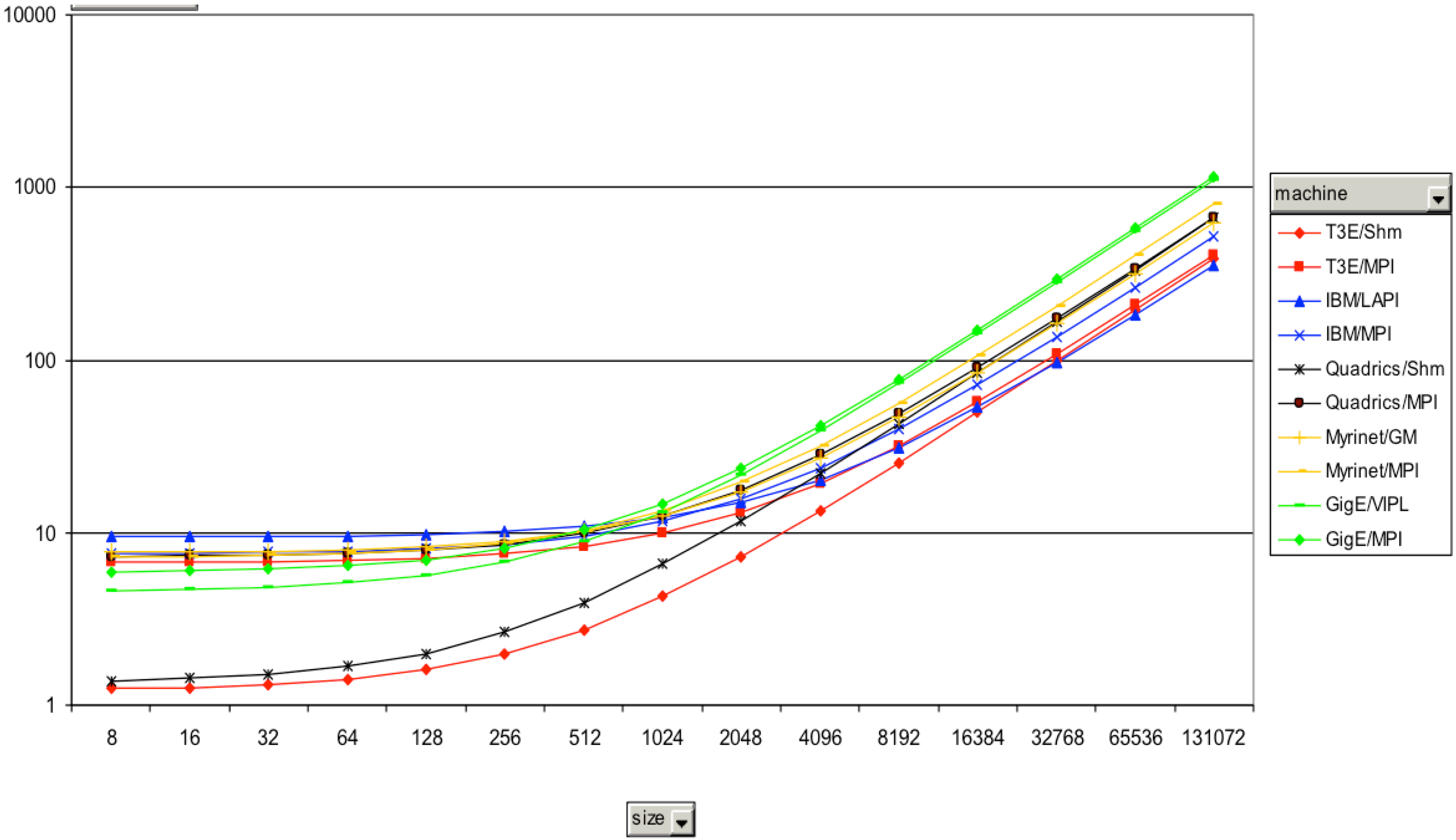
α is latency in usecs
β is BW in usecs per Byte

How well does the model

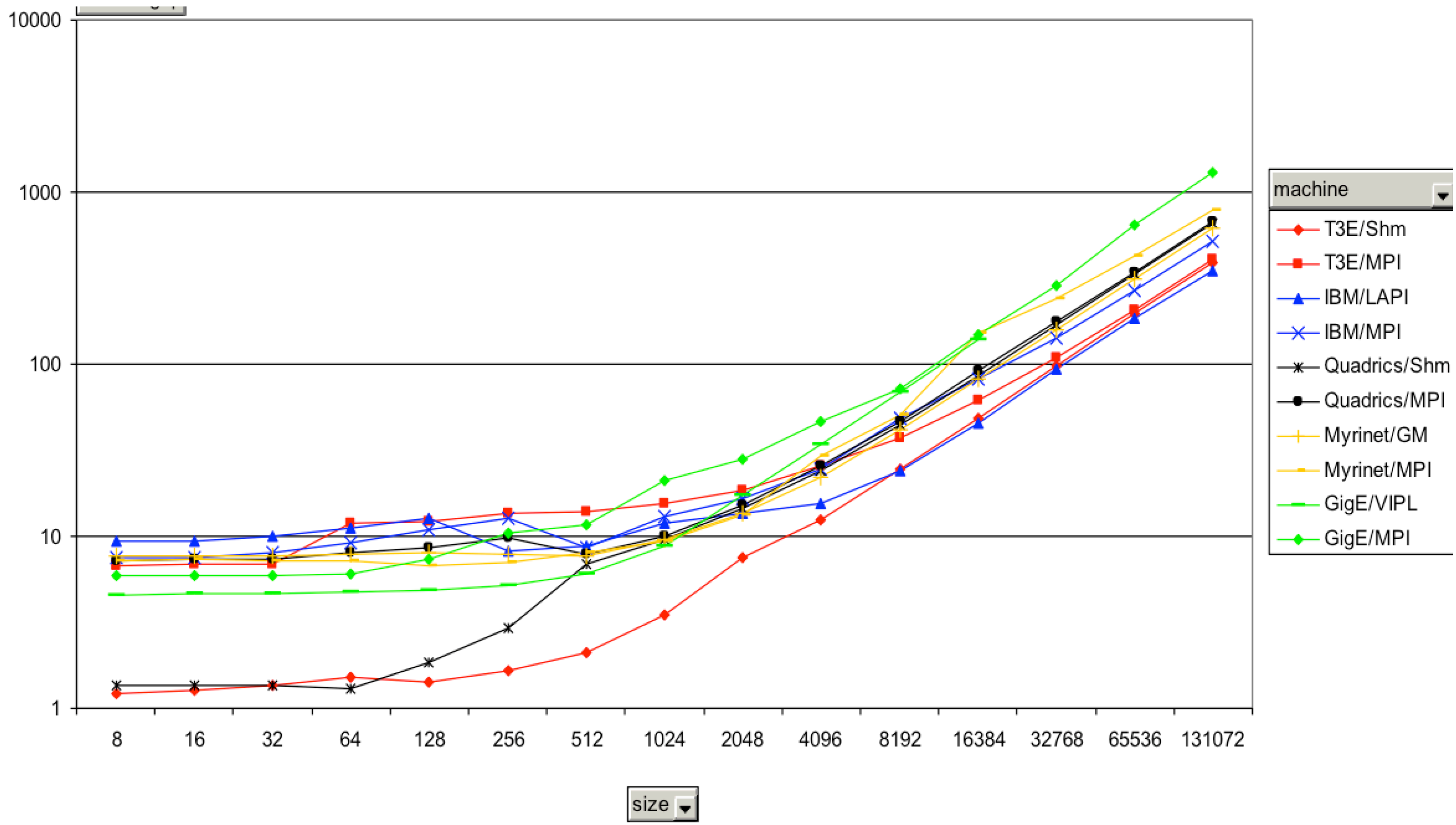
Time = α + n*β

predict actual performance?

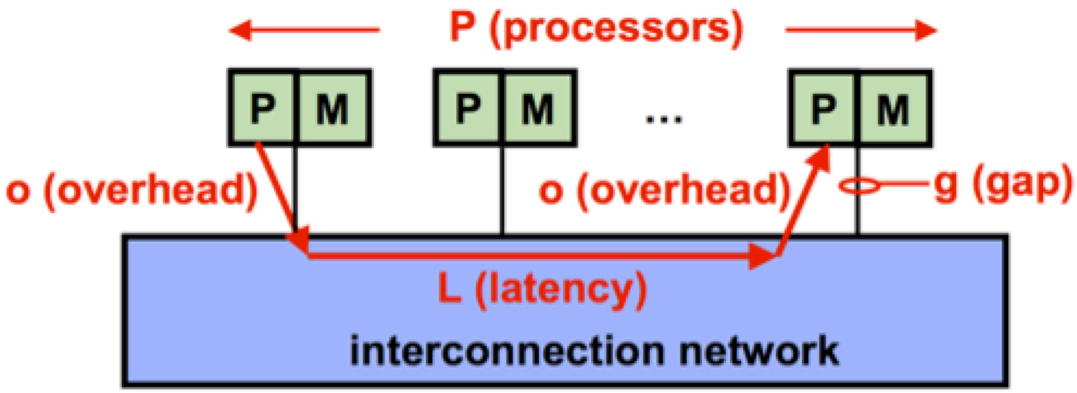
Model Time Varying Message Size & Machines



Measured Message Time



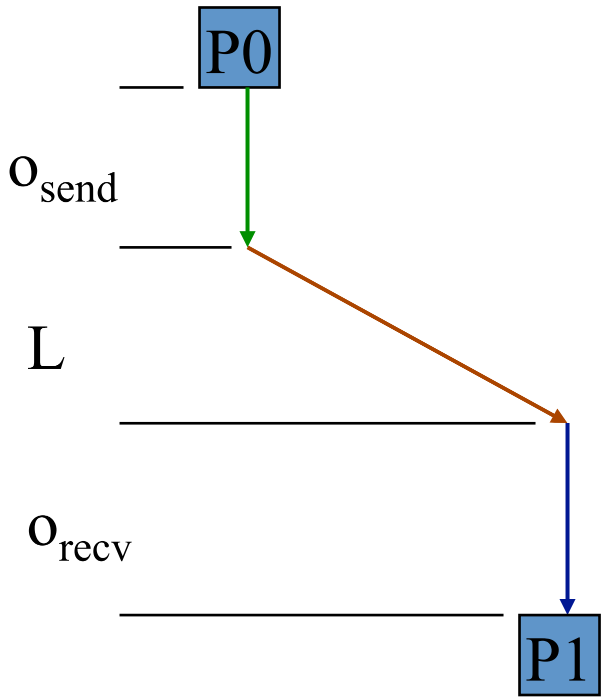
LogP Model



- 4 performance parameters
  - L: latency experienced in each communication event
    - time to communicate word or small # of words
  - o: send/recv overhead experienced by processor
    - time processor fully engaged in transmission or reception
  - g: gap between successive sends or recvs by a processor
    - 1/g = communication bandwidth
  - P: number of processor/memory modules

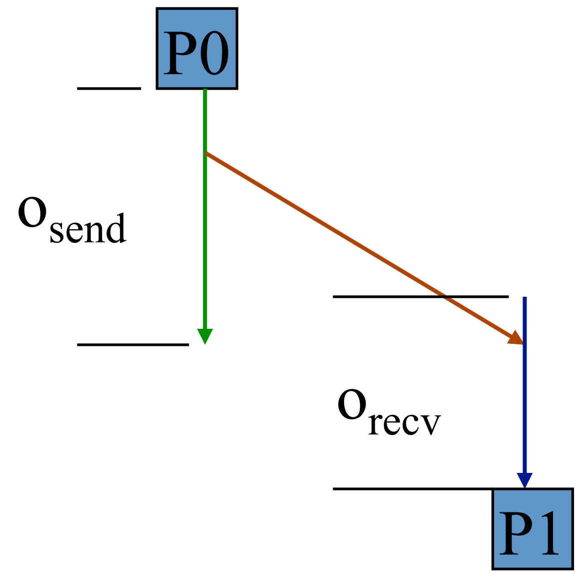
LogP Parameters: Overhead & Latency

- Non-overlapping overhead



$\mathbf{EEL = End\text{-}to\text{-}End\ Latency}$
$\mathbf{= o_{send} + L + o_{recv}}$

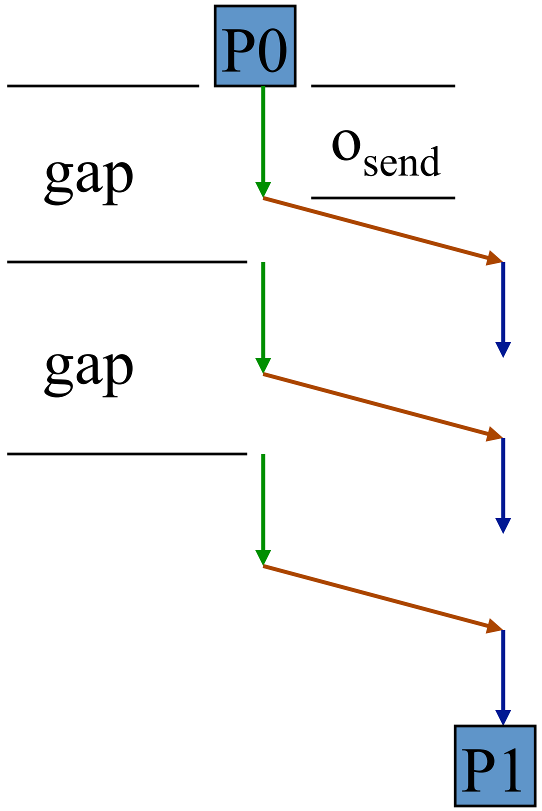
- Send and recv overhead can overlap



$\mathbf{EEL = f(o_{send}, L, o_{recv})}$
$\mathbf{\geq max(o_{send}, L, o_{recv})}$

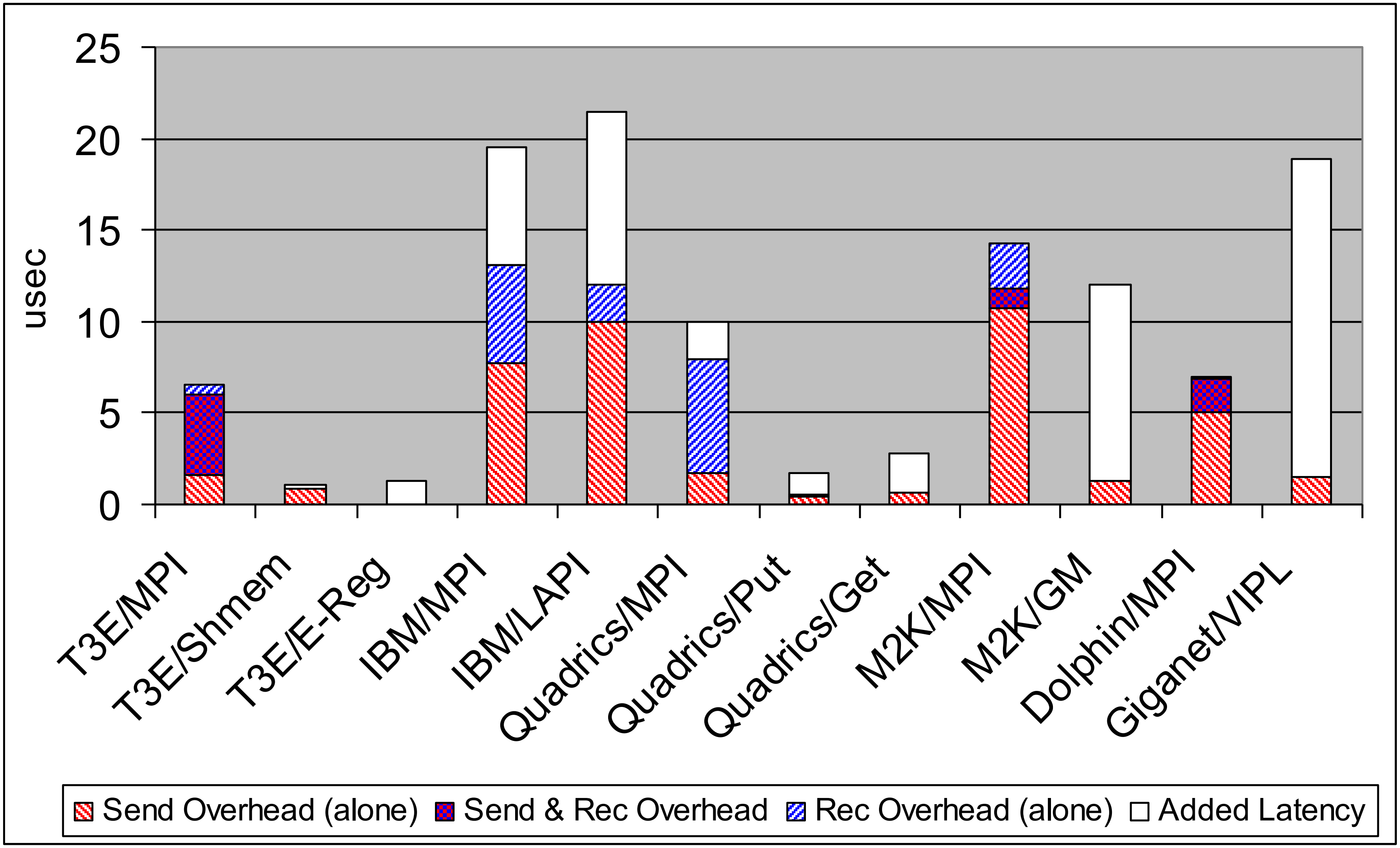
LogP Parameters: gap

- The Gap is the delay between sending messages
- Gap could be greater than send overhead
  - NIC may be busy finishing the processing of last message and cannot accept a new one.
  - Flow control or backpressure on the network may prevent the NIC from accepting the next message to send.
- No overlap ⇒ time to send n messages (pipelined) =



$$(o_{send} + L + o_{recv} - gap) + n*gap = \alpha + n*\beta$$

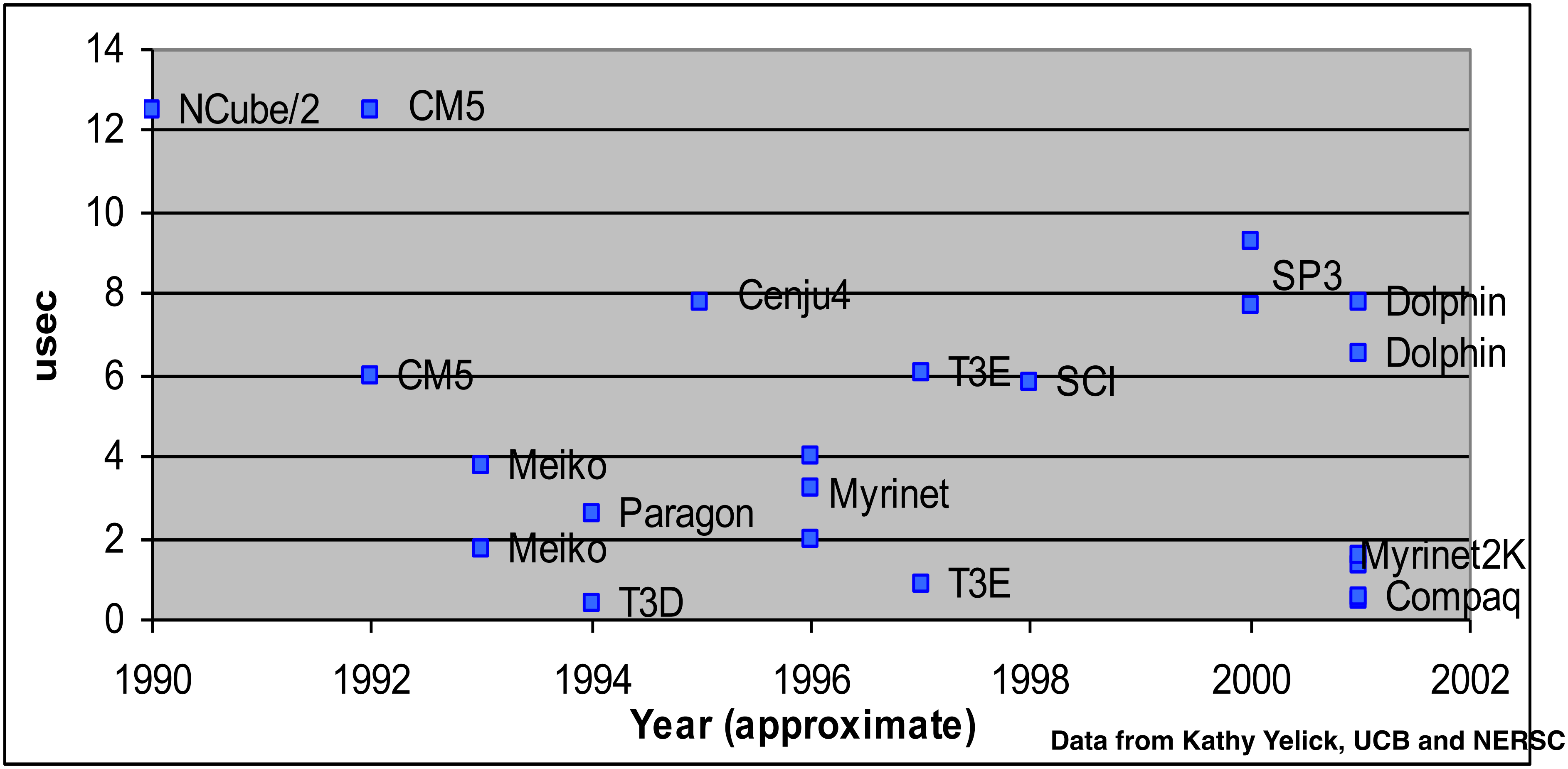
Results: EEL and Overhead



Data from Mike Welcome, NERSC

Send Overhead Over Time

- Overhead has not improved significantly; T3D was best
  - Lack of integration; lack of attention in software



Limitations of the LogP Model

- The LogP model has a fixed cost for each message
  - This is useful in showing how to quickly broadcast a single word
  - Other examples also in the LogP papers
- For larger messages, there is a variation LogGP
  - Two gap parameters, one for small and one for large messages
  - The large message gap is the b in our previous model
- No topology considerations (including no limits for bisection bandwidth)
  - Assumes a fully connected network
  - OK for some algorithms with nearest neighbor communication, but with "all-to-all" communication we need to refine this further
- This is a flat model, i.e., each processor is connected to the network
  - Clusters of multicores are not accurately modeled

Summary

- Latency and bandwidth are two important network metrics
  - Latency matters more for small messages than bandwidth
  - Bandwidth matters more for large messages than bandwidth
  - **Time = α + n*β**
- Communication has overhead from both sending and receiving end
  - **EEL = End-to-End Latency = $o_{send} + L + o_{recv}$**
- Multiple communication can overlap

Outline

- Cluster Introduction
- Distributed Memory Architectures
  - Properties of communication networks
  - Topologies
  - Performance models
- ☛ Programming Distributed Memory Machines using Message Passing
  - Overview of MPI
  - Basic send/receive use
  - Non-blocking communication
  - Collectives

Programming With MPI

- MPI is a library
  - All operations are performed with routine calls
  - Basic definitions in
    - mpi.h for C
    - mpif.h for Fortran 77 and 90
    - MPI module for Fortran 90 (optional)

Slide source: Bill Gropp, ANL

MPI: the Message Passing Interface

The minimal set of MPI routines.

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of calling process.
`MPI_Send`	Sends a “unbuffered/blocking” message.
`MPI_Recv`	Receives a “unbuffered/blocking message.

It is possible to write fully-functional message-passing programs by **using only the six routines**.

SPMD Program Models

- SPMD (Single Program, Multiple Data) for parallel regions
  - All PEs (Processor Elements) execute the same program in parallel, but has its own data
  - Each PE uses a unique ID to access its portion of data
  - Different PEs can follow different paths through the same code
- Each PE knows its own ID

```
if(my_id == n) {   }
else {    }
```

- SPMD is by far the most commonly used pattern for structuring parallel programs
  - MPI, OpenMP, CUDA, etc

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **MPI_Comm_size** reports the number of processes, size
  - **MPI_Comm_rank** reports the *rank*, a number between 0 and size-1, identifying the calling process

Slide source: Bill Gropp, ANL

Hello World (C)

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Try this on login.secs.oakland.edu

```
mpicc mpihello.c
mpd&
mpirun -np 4 ./a.out
```

Notes on Hello World

- All MPI programs begin with **MPI_Init** and end with **MPI_Finalize**
- **MPI_COMM_WORLD** is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI "job"
- Each statement executes independently in each process
  - including the `printf/print` statements
- Libc I/O is NOT part of MPI-2
  - print and write to standard output or error not part of either MPI-1 or MPI-2
  - output order is undefined (may be interleaved by character, line, or blocks of characters),
- To run with 4 processes

```
mpirun -np 4 a.out
```

Slide source: Bill Gropp, ANL

Some Basic Concepts

- Processes can be collected into **groups**
- Each message is sent in a **context**, and must be received in the same context
  - Provides necessary support for libraries
- A group and context together form a **communicator**
- A process is identified by its **rank** in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**

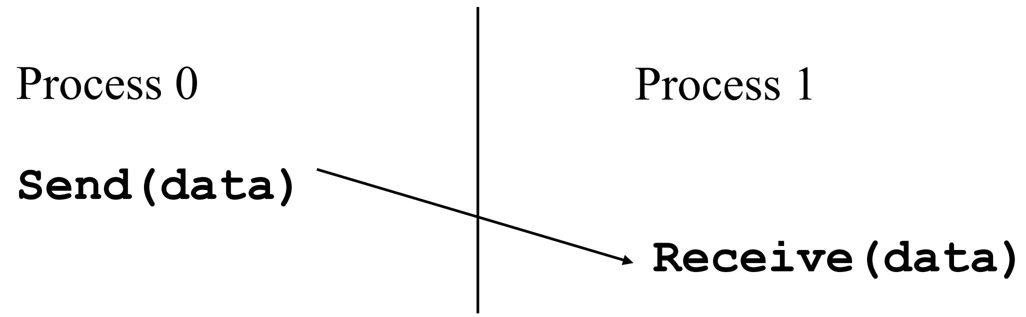
Slide source: Bill Gropp, ANL

Communicators

- A communicator defines a *communication domain*
  - A set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type **`MPI_Comm`**.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different (possibly overlapping) communication domains.
- **`MPI_COMM_WORLD`** includes all the processes.

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

Slide source: Bill Gropp, ANL

MPI Datatypes

- The data in a message to send or receive is described by a triple (**address, count, datatype**), where
- An MPI datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, in particular ones for subarrays
- May hurt performance if datatypes are complex

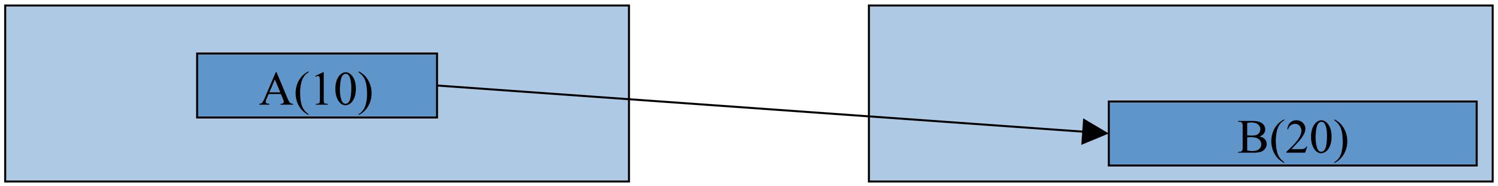
Slide source: Bill Gropp, ANL

MPI Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying MPI_ANY_TAG as the tag in a receive
- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes

Slide source: Bill Gropp, ANL

MPI Basic (Blocking) Send



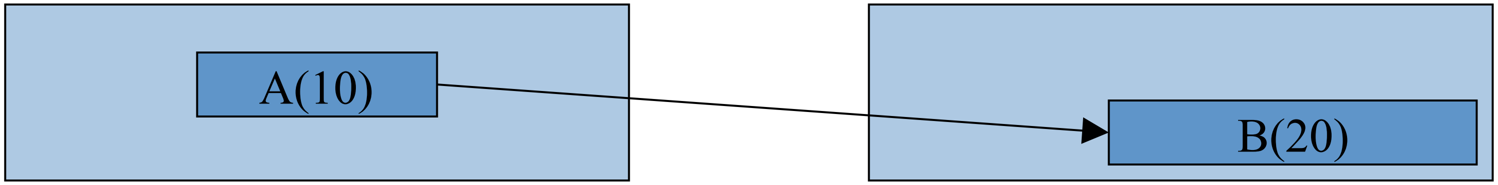
`MPI_Send( A, 10, MPI_DOUBLE, 1, …)` `MPI_Recv( B, 20, MPI_DOUBLE, 0, … )`

`MPI_SEND(start, count, datatype, dest, tag, comm)`

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

Slide source: Bill Gropp, ANL

MPI Basic (Blocking) Send



`MPI_Send( A, 10, MPI_DOUBLE, 1, …)` `MPI_Recv( B, 20, MPI_DOUBLE, 0, … )`

`MPI_RECV(start, count, datatype, source, tag, comm, status)`

- Waits until a matching (both **`source`** and **`tag`**) message is received from the system, and the buffer can be used
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**
- **`tag`** is a tag to be matched on or **`MPI_ANY_TAG`**
- receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error
- **`status`** contains further information (e.g. size of message)

Slide source: Bill Gropp, ANL

A Simple MPI Program

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[])
{
  int rank, buf;
  MPI_Status status;
  MPI_Init(&argv, &argc);
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  /* Process 0 sends and Process 1 receives */
  if (rank == 0) {
    buf = 123456;
    MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  } else if (rank == 1) {
    MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
          &status );
    printf( "Received %d\n", buf );
  }

  MPI_Finalize();
  return 0;
}
```

SPMD Model

Slide source: Bill Gropp, ANL

Retrieving Further Information

- `Status` is a data structure allocated in the user's program.

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

Slide source: Bill Gropp, ANL

Tags and Contexts

- Separation of messages used to be accomplished by use of tags, but
  - this requires libraries to be aware of tags used by other libraries.
  - this can be defeated by use of "wild card" tags.
- Contexts are different from tags
  - no wild cards allowed
  - allocated dynamically by the system when a library sets up a communicator for its own use.
- User-defined tags still provided in MPI for user convenience in organizing application

Slide source: Bill Gropp, ANL

Running MPI Programs

- The MPI-1 Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.
- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.
- **`mpiexec <args>`** is part of MPI-2, as a recommendation, but not a requirement, for implementors.
- Use

    mpirun –np # -nolocal a.out

  for your clusters, e.g.

    mpirun –np 3 –nolocal cpi

Slide source: Bill Gropp, ANL

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_SEND`
  - `MPI_RECV`

Slide source: Bill Gropp, ANL

The three examples

- Send/receive
- Ping-poing
- Ring

MPI References

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - http://www.mcs.anl.gov/mpi
    - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages
  - https://computing.llnl.gov/tutorials/mpi/

Slide source: Bill Gropp, ANL

Books on MPI

- *Using MPI: Portable Parallel Programming with the Message-Passing Interface (2nd edition)*, by Gropp, Lusk, and Skjellum, MIT Press, 1999.
- *Using MPI-2: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Thakur, MIT Press, 1999.
- *MPI: The Complete Reference - Vol 1 The MPI Core,* by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1998.
- *MPI: The Complete Reference - Vol 2 The MPI Extensions*, by Gropp, Huss-Lederman, Lumsdaine, Lusk, Nitzberg, Saphir, and Snir, MIT Press, 1998.
- *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.
- *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.